A Service Based Approach for a Cross Domain Reference Architecture Development

Liliana Dobrica[1] and Eila Ovaska[2]

[1] University Politehnica of Bucharest, Faculty of Automatic Control and Computers
Spl. Independentei 313, Romania

[2] VTT Technical Research Centre of Finland, Oulu, Kaitoyvala 1, Finland

Abstract. One trend of software engineering is that systems are in transition from component based architectures towards service centric ones. Also techniques from software product lines can help in a quality based and systematic reuse. The content of this paper addresses the issue of how to perform design and quality analysis of cross domain reference architecture. The reference architecture is designed based on the domains requirements and features modelling. We propose a service based approach for cross-domain reference architecture development. Throughout the sections we try to introduce an innovative way of thinking founded on bridging concepts from software architecture, service orientation, product line and quality analysis with the purpose to initiate and evolve software systems products.

1 Introduction

In software development domain systems of yesterday become components of today. The fundamental principle stating that "any system consists of components" is common for any technical system and it is sometimes called "a law of nature" [6]. Among the requirements and constraints that have to be satisfied we can mention a higher diversity and complexity of systems and components, increased quality, productivity and reuse content, standardization, stricter requirements for time-to-market. The domain technology causes exponential growth of the designed systems.

Nowadays many systems are used as subsystems in a variety of domains such as enterprise systems, embedded systems, and so on. In these domains there is a variety of functions; however they might be composed of a limited number of common software/hardware components. Nowadays in various industries it has been recognized a significant duplication of development effort for hardware, software and services [1]. Due to the escalating complexity level, the technology trends and the bigger competition in the world market, a coherent and integrated development strategy is required. It becomes a research priority the creation of a generic platform and a suite of abstract components with which new developments in different application domains can be engineered with minimal effort. Generic platforms, or reference designs, can be based on a common architectural style that supports the

Dobrica L. and Ovaska E. (2009).
A Service Based Approach for a Cross Domain Reference Architecture Development.
In *Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering - Evaluation of Novel Approaches to Software Engineering*, pages 33-44
DOI: 10.5220/0001969600330044
Copyright © SciTePress

composition of systems out of pre-validated independently developed subsystems that meet the requirements of the different application domains. Given a core architectural style, different components are created for different application domains, while retaining the capability of component reuse across these domains.

Reference architecture (RA) serves several purposes, of which the most important are knowledge base, starting point and reuse. Knowledge base represents a common terminology for software system architects. The shared terminology enables architects to share experiences more efficiently. Starting point means that architectural documentation can be used as a starting point for an iterative development process, reducing in this way the effort for designing architectures for new products. Reuse is in the sense that the RA describes the generic structure and behavior of the services. This makes integrating existing “compliant” software components easier and thus increases the reuse potential of those services. The RA functionality, interfaces and constraints are abstract and complex. Not all the development organizations will understand them well. Not knowing RA capabilities may lead to the architecture not being fully used. However, the aim is that all products should fit into the provided architecture and benefit from it. Requirements that have already been considered might be re-implemented for various products. An impact of multi-implemented requirements could be an unstable RA.

In this paper we propose a coherent and integrated development strategy for complex systems that considers the architecture the main driver. We argue with our experiences in the software architectures design and analysis for various domains [4, 5] and other researchers’ recent studies that will be revealed during the paper. Our contribution is in the synthesis of the most important issues that can be applied in a cross domain development strategy based on quality. We propose a service based approach for cross-domain RA development.

2 Background

2.1 Software and Service Architecture

Software architecture (SA) provides design-level models and guidelines for composing software systems. The SA is defined as “the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them” [12]. The SA description is designed to address the different perspectives one could have on the architecture. Each perspective is a view. The information relevant to one view is different from that of others and should be described using the most appropriate technique. Several models have been proposed that include a number of views that should be described in the software architecture. The view models have something in common, and that is that they address the static structure, the dynamic aspect, the physical layout and the development of the system. In general, it is the responsibility of the architect to decide which view to use for describing the SA. Architectural styles are recurring patterns of system organization whose application results in systems with known, desirable properties. In practice, an architectural style consists of rules and guidelines for the partitioning of a system into subsystems and for the design of the interactions among subsystems. The subsystems must comply with the architectural style to avoid a property mismatch at the interfaces between subsystems.

Service architecture is a set of concepts and principles for specification, design, implementation and management of software services [7]. This definition is similar to SA that also includes the principles for guiding its design and evolution and has a strong influence over the lifecycle of a system [10]. Service architecture refers mostly to the software architecture of applications and middleware which is the software that is located between applications and network layer. A middleware layer hide the underlying network environment complexity insulating applications from explicit protocol handling, disjoint memories, data replication and parallelism. Furthermore, the middleware layer masks the heterogeneity of operating systems, programming languages and networking technologies to facilitate application programming and management [8]. A service based approach provides support for adaptability and flexibility of components and frameworks [9]. A design approach of services at architectural level has to consider quality attributes and standards.

2.2 The Software Product Line Development

In general the software product line development consists of two stages which are domain engineering and application engineering [15]. Domain engineering is divided in: Domain Analysis, Domain Design and Domain Implementation. The domain analysis consists in capturing information and organizing it as a model. Some methods, such as FODA (Feature-Oriented Domain Analysis) [3] propose a set of notations for the domain modeling using the notion of "features" to refer to products properties. The input represents domain knowledge and outputs are domain requirements. The domain design consists in establishing the product line architecture. The domain implementation consists of implementing the architecture defined during the domain design as software components. The results represent core assets such as, domain requirements, product-line architecture and components. The application engineering stage consists in building products based on the results of domain engineering and users needs. During application analysis of a new system, the requirements from the existing domain model, which matches the user’s needs, are selected. Applications are assembled from the existing reusable components. Variability management is a key issue in the success of product line development.

2.3 Quality Evaluation Techniques at the Architectural Level. Scenarios

Evaluation techniques are categorized in questioning and measuring techniques [12]. The first category generates qualitative questions to ask about a SA and they are applied to evaluate SA for any given quality. Questioning techniques include scenarios, questionnaires and checklists. Measuring techniques suggest quantitative measurements to be made on SA. They are used to answer specific questions and to address specific software qualities, and therefore they are not as broadly applicable as questioning techniques. This category includes metrics, simulations, prototypes and experiences. In terms of quantitative and qualitative aspects, both classes of techniques are needed for evaluating SA. Various analyzing models expressed in formal methods are included in quantitative techniques. Qualitative techniques illustrate SA evaluations using scenarios. Scenarios are rough, qualitative evaluations of architecture. Scenarios are necessary but not sufficient to predict and control quality attributes and have to be supplemented with other evaluation techniques. Including questions about quality indicators in the scenarios enriches SA evaluation.

The existing practices with scenarios are systematized in [12]. The usage of scenarios is motivated by the consensus it brings to understanding of what a particular software quality really means. Scenarios are a good way of synthesizing individual interpretations of a software quality into a common view. This view is more concrete than the general software quality definition and it also incorporates the specifics of a system to be developed, i.e. it is more context sensitive. Scenarios are a postulated set of uses or modifications of the system and they are typically one sentence long and the modifications reflected in scenarios could be a change to how one or more components perform an assigned activity, the addition of a component to perform some activity, the addition of a connection between existing components, or a combination of these factors. The scenario development is based on the system requirements that are considered in the architecture. Scenarios have to be sufficiently concrete to ensure the expressiveness of the analysis.

3 Our Approach

3.1 Architecture Design

We define a cross domain approach that extends to three levels the architecture development of a software system (Fig. 1.). We consider the system as a collection of cooperating services that deliver required functionality. These services may be executed in a networked environment and may be recomposed dynamically. The RA level includes core services and focuses on commonality analysis. Also the RA includes rules or constraints on how core services should be combined to realize a particular functional goal. Domain architecture level includes domain specific services and requires variability management concerns. The last level is dedicated to the set of product architectures, where rules for product derivation and configuration are included. A feature model is a prerequisite of our approach [2]. This model is essential for both variability management and product derivation, because it describes the requirements in terms of commonality and variability, as well as defining dependencies. We have built a UML meta-model for features modelling (Fig. 2.). The features model specifies dependencies called composition rules. The *requires rule* expresses the presence implication of two features and the *mutually exclusive rule* captures the mutual exclusion constraint on feature combinations.

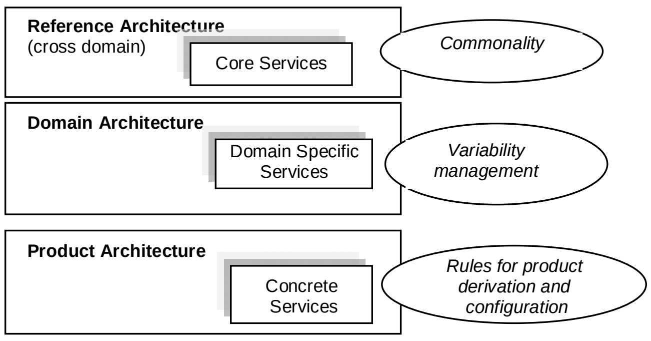

Fig. 1. Architecture development approach.

RA defines quality attributes, architectural styles and patterns and abstract architectural models (Fig. 3.). *Quality attributes* clarify their meaning and importance for core service components. The interest of the quality attributes for the RA is how the quality attribute interacts with and constrains the achievement of other quality attributes. Services have to meet many quality attributes. Modifiability of a service is divided into the ability to support new features, simplify the functionality of an existing system, adapt to new operating environments, or restructure system services. Integrability measures the ability of the parts of a system to work together. It depends on the external complexity of the components, their interaction mechanisms and protocols, and the degree to which responsibilities have been cleanly partitioned.

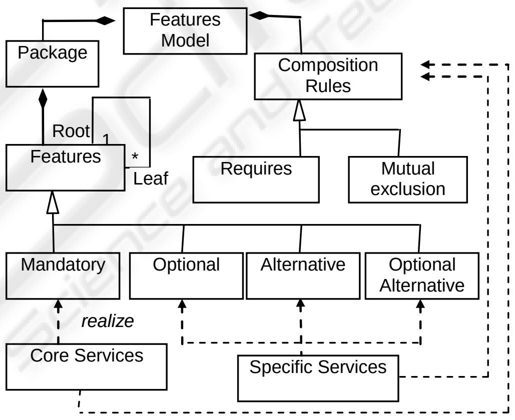

Fig. 2. Features - UML metamodel.

The *styles and patterns* are the starting point for architecture development. Architectural styles and patterns are means to achieve qualities. A style defines a class of architectures and is an abstraction for a set of architectures that meet it. An architectural pattern is a documented description of a style or a set of styles that expresses a fundamental structural organization schema applied to high-level system subdivision, distribution, interaction, and adaptation [13].

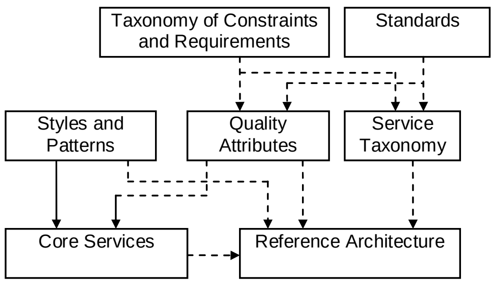

Fig. 3. RA realization.

Design patterns, on the other hand, are on a detailed level. They refine single components and their relationships in a particular context [14]. In this way the RA creates the framework from which the architecture of new products is developed. It provides generic architectural services and imposes an architectural style for constraining specific domain services in such a way that the final product is understandable, maintainable, extensible, and can be built cost-effectively. Potential reusability is highest on RA level. Core services and the architectural style of the RA are reused in every domain architecture. RA is build based on a *service taxonomy*. We adopted the idea from WISA [11] of an existing knowledge on software engineering that is integrated and adapted to service engineering. The standards related to each domain, applicable styles and patterns and existing concepts of services and components are the driving forces in system development. A service taxonomy defines the main categories called domains. Typical features that have been abstracted from requirements characterize services. The reason of service taxonomy is to guide the developers on a certain domain and getting assistance in identifying the required supporting services and features of services.

Domain architecture describes ready made building blocks that assist application/products developers in using specific domains services. When the RA has been defined, the existing components and services are considered as building blocks in the architecture of the set of products. The domain services provides variable assets repository. Variability appears in functional and non-functional requirements (including quality attributes). A structured domain architecture repository may be provided at this level. A schema for this repository has to be defined in a form of relationships between services. In this way we are mapping domain specific services to core abstract services. Specialization relation is a solution to be used for variability management. Run-time quality attributes variability requires tool support for modelling. The tool must provide monitoring mechanisms, measuring techniques and decision models for making tradeoffs [13].

Product architecture level consists of concrete services derived and configured based on rules. The goal of product derivation is to reach a configuration in which necessary variabilites have been bound. The decision model for bounding specific services of a domain to a product may be in a tabular form or a more comprehensive tool based on the feature types and composition rules. By selecting a consistent set of features required for the individual product, the corresponding domain specific services that realize those features are selected from the domain architecture repository to constitute the product.

3.2 Architecture Analysis

We have applied an analysis method that consists of the following five steps:

1. Deriving of change categories from the problem domain. Fig. 4. presents five categories of the change scenarios derived from the problem domains. A change scenario related to one of these categories may require other changes in the other categories. It is recommended to consider this possibility in the scenario development process. Usually it occurs when the problem domain is organized so that it is easy to identify the main sources for the addition of subsequent features in the domain.

2. Scenarios identification. Possible changes may happen in the life of the system based on the derived categories. Scenarios should illustrate the kinds of anticipated changes that will be made to the system. A common problem of the scenario development is when to stop generating scenarios. Using a set of *standard quality attribute-specific questions* we ensure proper coverage of an attribute by the scenarios. The boundary conditions should be covered. A standard set of quality-specific questions allows the possibility of extracting the information needed to analyze that quality in a predictable, repeatable fashion. The architecture is a good one and it is not necessary to generate scenarios to verify the functional requirements. Otherwise these should also be considered when verifying functionality. For analyzing the modifiability we must look for possible changes in the problem domain.

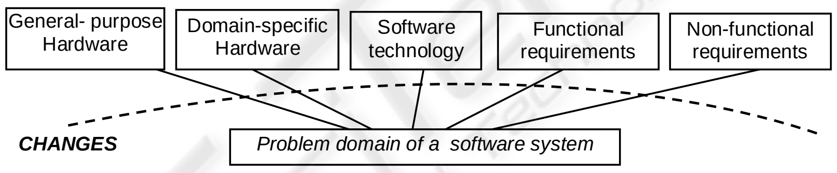
General- purpose Hardware
Domain-specific Hardware
Software technology
Functional requirements
Non-functional requirements
CHANGES
Problem domain of a software system

Fig. 4. Categories of scenarios.

3. Architecture Description could be performed in parallel with the previous one. Architecture description may use multiple views. For a common level of understanding a small and simple lexicon could be used in describing structures.

4. Evaluate the effect of the scenarios on the architecture elements. The effect is estimated by investigating which services are affected by that scenario. The cost of the modifications associated with each change scenario is predicted by listing the services that are affected and counting the number of changes. The objective is to get a measurement of the quality of the core and domain services with respect to the anticipated variability in functional or non-functional characteristics.

5. Scenario interaction. The result of the effects evaluation represents the input for this step. The activity is to determine which scenarios affect the same service. High interactions of unrelated scenarios indicate a poor separation of concerns. If any of the scenarios affect a core service this is no more part of the RA, but a domain specific.

4 Example

4.1 Example Description

Our example is abstracted from our experiences with the architecture design of a scientific on-board silicon X-ray array (SIXA) spectrometer control software. SIXA is a multi-element X-ray photon counting spectrometer. It consists of specific domain hardware elements. The SIXA measurement activity consists of observations of time-resolved X-ray spectra for a variety of astronomical objects. Fig. 5 introduces the context view of SIXA considering it a measurement controller. External elements are a command interface and physical devices, i.e. sensors and actuators. The system is programmed and operates using a set of commands sent from a command interface.

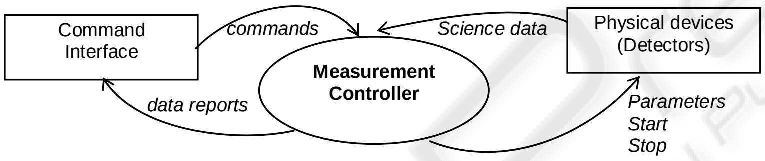

Fig. 5. Context view of the system.

The role of the spectrometer controller is to control the following modes: (a) Energy Spectrum (EGY), which consists of three energy-spectrum observing modes: Energy-Spectrum Mode (ESM), Window Counting Mode (WCM) and Time-Interval Mode (TIM). (b) SEC, which consists of single event characterization observing modes: SEC1, SEC2 and SEC3. Each mode could be controlled individually. A coordinated control of the analog electronics is required when both measurement modes are on.

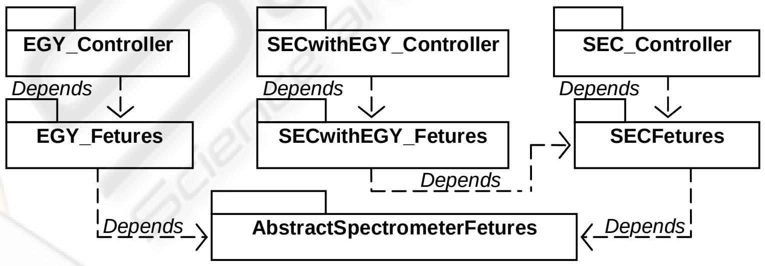

Fig. 6. Mapping features into packages.

The analysis result of requirements for domain engineering is the features model. This has been structured in packages (Fig. 6.). The *with reuse* aspect of reusability is described in the architecture by the abstract features. The abstract features encapsulated in three main abstract domains MeasurementController, Data Management and DataAcquisition, are completely reused in all the derived products. The AbstractSpectrometerFeatures package has the highest degree of reusability but also the highest degree of dependability. The abstract features depend on the commonality between EGY and SEC features. A change in the problem domain of a product is reflected in the degree of reusability of the abstract domain features.

The sets of products that could be derived from the domain specific services during application engineering are: (1) P1 – EGYController includes specific services of a standalone control of EGY mode; (2) P2 – SECController includes specific services of a standalone control of SEC mode; (3) P3 – SECwithEGYController includes specific services of coordinated control.

4.2 Example Architectural Design

The architecture model is documented around multiple views describing conceptual and concrete levels, for each view a static and dynamic perspective being offered. Architecture documentation addresses specific concerns for measurement control, data acquisition control and data management. The views are illustrated with diagrams expressed in UML-RT, a real-time extension of UML.

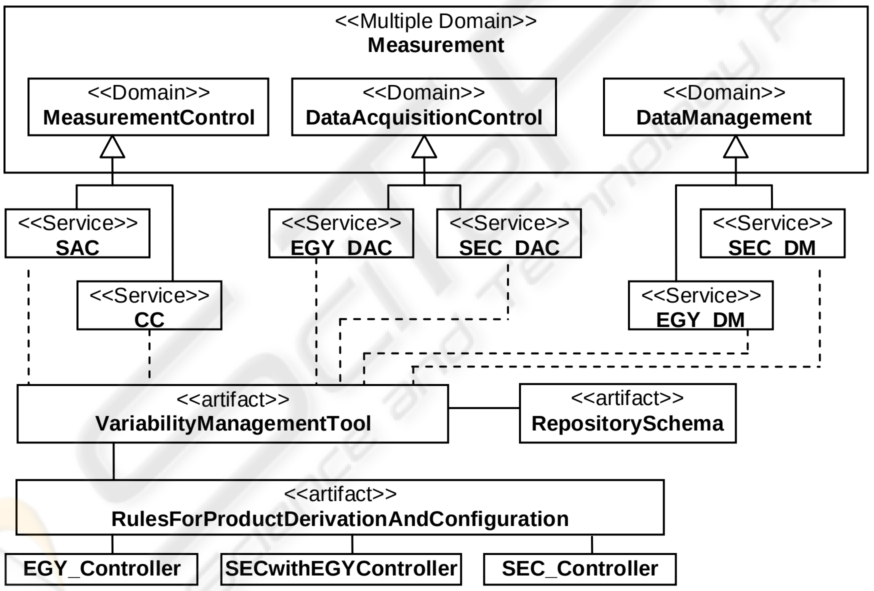

Fig. 7. Spectrometer controller cross domain architecture design approach.

The conceptual level considers a functional decomposition of the architecture into domains. The relationships between architectural elements are based on pass control and pass data or uses. The concrete level considers a more detailed functional description, where the main architectural elements are packages, capsules, ports, protocols. The relationships are association, specialization, generalization, etc. Considering the dynamic aspect statechart diagrams and message-sequence charts are also part of this description level. Fig. 7 presents the spectrometer controller cross domain architecture design approach. The RA encapsulated in the Measurement «Domain» is composed of three core abstract «Domain»s Measurement Control, DataAcquisitionControl and DataManagement. In each core «Domain» abstract features are collected. The MeasurementControl is responsible for services of starting and stopping the operating mode for data acquisition according to the commands received from the command interface and according to the events generated in other parts of the software. DataAcquisitionControl service collects events (science data) to the spectra data file during observation of a target. This abstract service includes as well as hides data acquisition details. DataManagement abstract services provide interfaces for storing science data, opening/closing/writing the data files, hiding storing details and controlling transmission of the stored data to command interface.

Domain architecture: Domain architecture consists of domain specific services and variability management services. Each of the three core services is specialized in domain specific services. For example, MeasurementControl is specialized in StandAloneControl (SAC) and CoordinatedControl (CC), DataAcquisitionControl (DAC) is specialized in EGY_DataAcquisitionControl (EGY_DAC) and SEC_DataAcquisitionControl (SEC_DAC), Data Management (DM) is specialized in EGY_Data Management (EGY_DM) and SEC_DataManagement (SEC_DM). This architecture includes services associated to variability management.

Table 1. Domain specific services and products.

Domain	Specific Service	Products		
		P1	P2	P3
MeasurementControl	SAC	x	x	
	CC			x
DataAcquisitionControl	EGY_DAC	x		x
	SEC_DAC		x	x
DataManagement	EGY_DM	x		x
	SEC_DM		x	x

Product architecture: Product architecture for the sets of products includes rules for product derivation and configuration. Table 1 presents domain specific services and products derivation. Products are horizontally distributed and the domain services are dispersed vertically. Each cell t_{ij} of the table is marked if product P_j uses component C_i. For example, two products, P1 and P2, include a SAC service of the measurement control domain.

4.3 Example Architecture Analysis

We have defined twelve change scenarios for changes in general purpose hardware, domain specific hardware, technology, functionality, non-functional requirements and other changes. For example a scenario in domain specific hardware category is: “Add a hard disk for SEC products.” Then we have analyzed the concrete structural view of the SA. A good SA design provides a good localization of changes. Most of the changes required by scenarios are applied to one component, which indicates a good decoupling of concerns. An important change is the addition of the hard disk, a variation among products. This scenario requires changes localized to specific domains services. By structuring the architecture in abstract domain services, which encapsulate common features of the multiple domains and domain specific services at a concrete level, which in turn represents specialization of the optional, alternative and optional alternative features, the effects of the change scenarios are minimized and localized. Changes did not affect the core services of the cross domain RA, which confirms the stability of the architecture across domains. The results of the analysis depend on the description of the architecture. By using only the decomposition view on the conceptual level the effects of the change scenarios are reduced because not all the details are included. On the concrete level, the views developed with the help of a CASE tool the effect of change scenarios is more relevant. This is an argument for that the evaluation method should be applied iteratively while the architecture design becomes more detailed. The purpose of the evaluation is to analyze the architecture to identify potential risks by predicting the quality of the products before they have been built. Iterative methods promote analysis at multiple resolutions as a means of minimizing risk at acceptable levels of time and effort. Areas of high risk are analyzed more profoundly (simulated, modeled or prototyped) than the rest of the architecture. Each iteration determines where to analyze more deeply in the next iteration.

The measurement controller domain also requires run-time qualities such as performance, safety and reliability. These are mandatory root features for the domain. However variants could include variability in these aspects. These variable features must be considered from the cross domain design perspective in order to minimize the risk that the final software products do not conform to these quality attributes. For architectural evaluation of these aspects several progresses have been identified in the literature that will be analyzed in our future work. It is important to estimate what is the degree of reuse at architectural level and what are the reusable assets when the variability of these run-time qualities is considered.

5 Conclusions

We have proposed an approach for software development based on a cross-domain RA. We have provided an incremental design and analysis approach based on services, which is more practical, easy to follow and benefits of advantages provided by service engineering. Our approach has been validated by a simple example. The problem dimension for the development of a cross-domain RA increases due to the larger number of requirements and constraints that may be specified by the complex systems domains. Building the features model may require a tool in order to manage the analysis and structuring the abstract features in domains. The cross domain RA contains core services of the domains included in the abstract features package. The appropriate architectural style is provided by a knowledge base through a service taxonomy. A domain architecture repository is a solution for variability management of specific services. A decision support tool is proposed for product derivation. The role of this tool is to bound variabilities in order to get a service configuration for a product architecture. In our example we developed a tabular form for the decision model. When the complexity increases a more elaborated tool is required and is a subject of our future research. The analysis strategy based on scenarios has been used to verify architecture against anticipated changes in domain knowledge. From the commonality viewpoint analysis results should consider if scenarios affect core services of the RA. If these core services are affected they should be domain specific.

Future research work is needed to develop systematic ways of bridging requirements taxonomy of each domain to a cross domain RA. However this paper presented the main concepts and justified why this concepts are required. When several domains adopt a service oriented approach it is possible to develop products which address functions from across two or more domains and consume services from multiple domains. Seeking engagement of communities of practice across domains is a more challenging but worthwhile goal. It remains to be seen as to how relevant international bodies foster such engagement. An essential prerequisite however is to have in place a coherent core services for each specific domain that can be used as a point of reference in establishing cross domain exchanges.

References

1. Kopetz H.: The ARTEMIS Cross-Domain Architecture for Embedded Systems, (2007)
2. Kang. K., S. Cohen, J. Hess, W. Novak, A. Peterson: Feature-Oriented Domain Analysis Feasibility Study, SEI Technical Report CMU/SEI- 90-TR-21 (1990)
3. Niemelä E., Evesti A., Savolainen, P, Modeling Quality Attribute Variability, Procs. of the 3rd Int. Conf ENASE., INSTICC Press, (2008) 169-176
4. Dobrica L., Niemelä E.: A survey on software architecture analysis methods, IEEE Trans. on Soft. Eng. Journal, 28(7), (2002) 638-653
5. Dobrica L., Niemelä E.: Modeling Variability in the Software Product Line Architecture of Distributed Services, Procs of SERP 2007, (2007) 269-275
6. Szypersky C.: Component Software Beyond Object-Oriented Programming, Addison-Wesley (1999)
7. TINA, Service Architecture Specification, http://www.tinac.com, (1997)
8. Dobrica L, Niemelä E: Adaptive middleware services, Procs. IASTED AI'2002, (2002)
9. Costa E., G. Blair, G. Coulson: Experiments with reflexive middleware, Procs. ECOOP'98 Workshop Reflexive Object Oriented Programming and Systems (1998)
10. IEEE Recommended Practice for Architectural descriptions of Software Intensive Systems, Std1 417-2000, (2000)
11. Niemelä E, Kalaoja J, Lago P: Towards an architectural knowledge base for wireless service engineering, IEEE Trans. on Soft. Eng., 31 (5), (2005) 361 – 379
12. Bass L., P. Clements, R. Kazman: Software Architecture in Practice, Addison-Wesley, (1998)
13. Buschmann F., R. Meunier, H. Rohnert: Pattern-Oriented Software Architecture:A System of Patterns, John Wiley and Sons, (1996)
14. Gamma E., R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, (1994)
15. Pohl K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Foundations, Principles, and Techniques. Springer-Verlag, (2005)